PROCESSING BUSINESS DATA AND STATISTICS FROM IBM WEBSPHERE COMMERCE

Pawel Giermek

Bachelor's Thesis
04.2012

Degree Programme in Information Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
Giermek, Pawel Marcin	Bachelor´s Thesis	18.04.2012
	Pages 45	Language English
	Confidential ( ) Until	Permission for web publication (X)

Title
PROCESSING BUSINESS DATA AND STATISTICS FROM IBM WEBSPHERE COMMERCE

Degree Programme
Software Engineering

Tutor(s)
Salmikangas, Esa

Assigned by
Descom Oy

Abstract
The objective of the thesis is to illustrate how business data can be retrieved and modified based on Groovy use. A variety of possibilities to extract and ways to change information required by a customer are presented.
First, the thesis presents simple examples to acquaint the reader with the topic. Tasks and changes made are more advanced and complex with time. To make it simple enough for people not familiar with SQL or WebSphere commerce application, examples are based on previous examples, thus it is only a matter of expanding the sample applications shown on the beginning.
The presented examples are fundamental: if a script is once written within Groovy aid, it can be reused any time it is needed for a similar task. Numerous samples that were listed during implementations prove the usefulness of developed applications.
The results shown are based on an existing application; however, not all presented data is authentic due to confidentiality issues. The conclusions chapter summarizes and proves the legitimacy of the topic: comparing additional examples based on previously shown working applications highlights all benefits for user.

Keywords
WebSphere, Groovy, database access, data retrieval, Java 2 Enterprise Edition, chart generating, HTML file generating, differences between Java and Groovy

Miscellaneous

CONTENTS

FIGURES

1 INTRODUCTION

The task of a support team in company working within e-commerce is to respond to the client and fix all issues that are discovered and reported. During that process, clients do not only notify about errors, but also often ask about different type of data, which is required for further development and future business plans. When responding to other support teams (responsible for finding bugs) it is reasonable to use professional language and raw data. Everything is different, when that will be shown at some point to officials and shareholders. Depending on that, the response must be appropriately selected. That is the main motivation for the choice of the subject of this thesis: to show how obtained information can be reported; what different ways can be used to properly present it and how it changed alongside with the gained experience. The context of this bachelor thesis is based upon practical training, which helped to get “know how” IBM schemas looks like, what kind of assignments can be expected and how differentiate the level of its difficulty.

An important fact is to divide clients: in this project, despite the fact that it is written based on WebSphere Commerce application, a client is not a customer who buys products on a website. A client is someone, who reports issues such as a not existing tag or a broken link, and usually works within a support team of a company using Commerce application.

An additional important objective was to make all scripts written in Groovy clear and universal, thus a little modification of task (like change of number URL link returned) would not cause necessity to diametrically change whole code. This is significant for a company, which supports many WebSphere Commerce applications, because it helps to save time, money, and helps to limit unnecessary work.

2 THEORETICAL BASIS

The aims of the thesis are to:

- Show the advantages provided by usage of Groovy
- Present how to retrieve data and different ways of processing it.

2.1 Welcome to the Groovy world

This whole paragraph is based on books about Groovy listed in references alongside with http://groovy.codehaus.org/. It is important to understand how it works and what connections it has to IBM implementations.

Groovy is an object-oriented programming language with syntax based on Java language (however it does not superset it). It is also type-safe, which means that an object of one type will not be (without conversion) treated as on object of the other type. It is as well open source, which makes it development easier. Setting it available for all kind of users fastens its development and makes community larger. Groovy runs on Java Virtual Machine and compiles to Java byte code – that allows ideal cooperation between Java and Groovy, however, there are some subtle differences between those two:

- Semicolons are optional
- Parentheses are optional
- The *return* keyword is optional
- Keyword *this* can be used inside static methods (that refers to this class)
- Default type of classes and methods is public
- Does not support inner classes
- The *throws* clause in a method signature is not checked by the Groovy compiler

- Groovy uses GroovyBeans (similar to JavaBeans) – properties of those look and act as public fields do, so it is not required to define explicit setters and getters[1]
- Passing arguments of not correct type or usage of undefined members will not cause compile errors.
- It is possible to overload whichever operator with own implementation
- Operator == denotes not identity of an object, but equality; for example: in Java: x.equals(y) is the same what x == y in Groovy.

Despite those, similarities are obvious, namely statements, exception handling or declaration of literals. What Groovy adds to this are assertions (used for documenting the code or to create self-checking code), which are more useful than comments or print statements - thanks to the fact they are always executed when code is.

Other good enhancements are closures – anonymous code blocks that are possible to declare outside of method or class (they will be executed only when called, not when defined). Generally, closure is assigned to a variable (it is identifier of this closure), which later is used to call on it. The real benefit that comes out of this is that it can be used anywhere in the program – thanks to that closures can be passed as arguments to other closures or methods.

What also comes with Groovy is support for collections (lists, maps and new structures: ranges). The most important difference between Java is that types of items belonging to one collection can be different without special declaration (it can be anything that is a subclass of *java.lang.Object)*:

- Java examples:
  - ✓ List<String> examp1 = **new** ArrayList<String> (Arrays.*asList*("John", "Doe", "male", "Chicago", "teacher"));

[1] except when there is need to change course or default behavior

- ✓ List<Object> examp2 = **new** ArrayList<Object> (Arrays.*asList*("John", "Doe", **false**, 22, 1));

- Groovy examples:
  - ✓ def examp1 = ['John', 'Doe', 'male', 'Chicago', 'teacher']
  - ✓ def examp2 = ['John', 'Doe', false, 22, 1]

Range is a sequence, which has a start and an end, as follows in the example below:

```
def examp = 1..5
```

Furthermore, Groovy adds:

- New classes, which helps with for example Extensible Markup Language (XML) processing or allows access to database
- Methods to an existing class of JDK (it can be done, because of object calls redirecting through its metaclass)
- New operators:
  - ✓ Searching: =~ (string on the left side will be used to create *Matcher* object; string on the right side will be pattern; in other words, it can be treated as two-dimensional array)
  - ✓ Matching: ==~ (depending on comparing string from left side to the pattern on the ride side appropriate Boolean will be returned)
  - ✓ For the regex pattern: ~Pattern (string will be compiled as regex pattern).

The aforementioned operators come in handy, when regexes (Regular Expressions) processing is concerned. Those are only extra features Groovy implements to make

work with regexes easier (other are: enhancement of *Pattern* and *Matcher* with additional methods and syntax that was simplified).

In addition, there are two types of string supported: default Java strings (java.lang.String) and those introduced by Groovy, GStrings (groovy.lang.GString); important is that GStrings are not extending String, because it is final. Instances of those can still be used as normal Strings – they are coerced into Java Strings. GStrings are especially useful, when strings have to be built dynamically.

2.2 WebSphere content

This chapter with all its content was based on IBM WebSphere V4.0 Advanced Edition Handbook – WebSphere commerce application is product of IBM that is why the most accurate information can be found in their tutorial book.

2.2.1 WebSphere introduction

The IBM WebSphere software platform is a set of solutions designed for e-business. It uses standards, which makes it easy to change and it also allows quick adaptation for the required purposes. Functionalities coming with it are listed below:

- Full Java 2 Enterprise Edition (J2EE) platform certification
- Tools for developing active Web sites through the use of Java servlets and JavaServer Pages (JSP). This functionality is available in all versions of the Advanced Edition
- Tight integration with tools for developing and deploying enterprise beans written to the EJB specification. Enterprise beans can act as a bridge between your Web site and your non-Web computer systems
- A set of application programming interfaces (APIs) for generating, validating, parsing, and presenting extensible markup language (XML) documents. This functionality is available in all versions of the Advanced Edition

- Integrated support for key Web services open standards, which is production-ready for the deployment of enterprise Web service solutions for interoperability and business-to-business applications
- A graphical user interface (GUI), the WebSphere Administrative Console, for administering the components of the Advanced Edition environment
- Unparalleled connectivity provided by a preview implementation of Java 2 Connectivity (J2C).
  (Compliete list taken from IBM WebSphere V 4.0 Advanced Edition Handbook)

Applications based on WebSphere are mostly integrated with Web clients (thin or thick) and can be integrated with (running in application servers) procedural applications. Within the application there are components responsible for fulfilling different objectives, like:

- persistence and query functions for enterprise beans are implemented by relational databases (new or existing ones)
- components are able to work together thanks to JavaBeans mechanisms
- access to databases, handling transactional operations and containing the application's business logic is carried out by enterprise beans
- program flow and user interface are provided by JSP and HTML pages
- Servlets can generate Web page contents dynamically; however, they are mainly responsible for coordinating work between the other components of the application.

2.2.2 Java 2 Enterprise Edition

J2EE is a definition of standard, which must be applied to all parts of architecting, developing and deploying applications that are server-based. J2EE architecture consists of the following elements: standard application model (responsible for developing applications), standard platform (applications hosting), compatibility test

suite (used for verification of harmony between products and platform standards) and reference implementation.

Specification of platform describes the runtime environment (includes components, resource manager drivers and containers – elements of this environment communicate with specified standard services) for a J2EE application. Figure 1 illustrates how components are distributed and what the server model looks like.

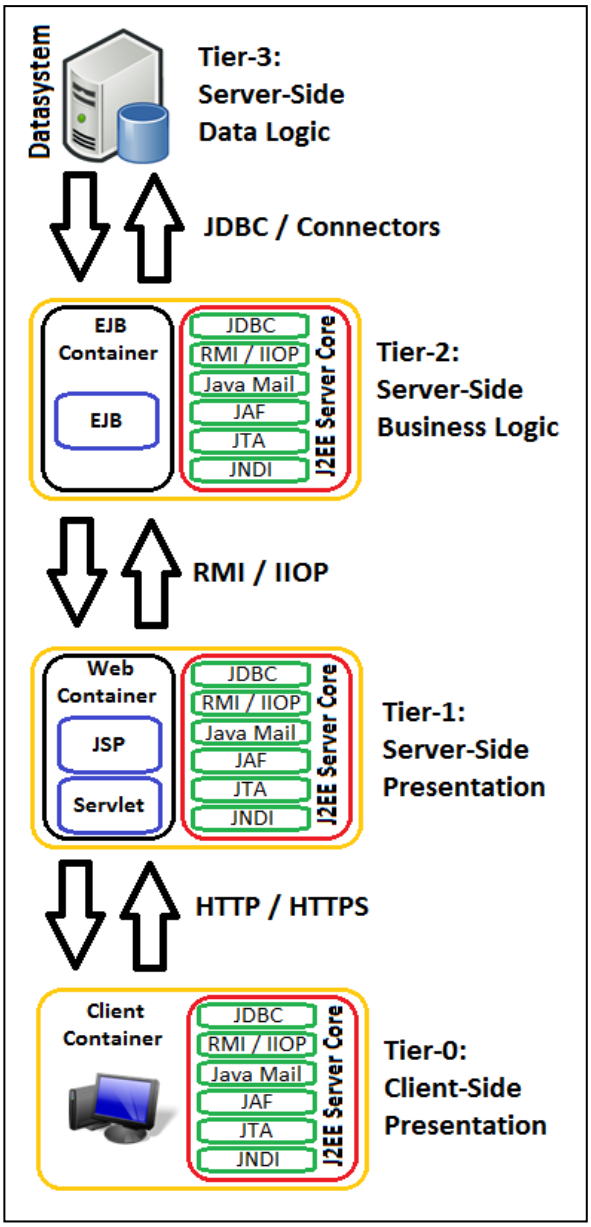

FIGURE 1: J2EE server model

There are also defined roles, which are performed through the development and deployment of application:

- development and packaging is possible thanks to the tools delivered by the tool provider
- application clients, applets, Web components, enterprise beans are created by application component provider
- Application assembler is responsible for assembling products of component provider into Enterprise Archive (EAR) file
- Enterprise application is deployed into an operational environment by deployer
- Environment in which the application runs is secured by system administrator.

The most significant benefits of Java 2 Enterprise Edition belong to the customers. Unified approach to securing application components, integration with legacy security schemas and single sign-on support are provided by elastic security model. Database connection pooling and distribution of containers across multiple systems improves scalability. The architecture is based on standards, which take advantage of Java run-anywhere technology. Vendors' unique solutions provide a wide choice of components and tools intended for application development. Reliable business transactions are achieved thanks to the services which provide integration with existing systems.

2.2.3 Foundation of J2EE

SUN defined Enterprise JavaBeans (EJB) specification and that is base for Java 2 Enterprise Edition (IBM WebSphere V4.0 Advanced Edition Handbook). It is used by

vendors during the process of an infrastructure implementation, which enables the components' deployment and usage of services (e.g. security, management of life-cycle or distributed transactions). Developer's work is limited to interpret specification properly and services reuse – that allows focusing on the business logic of the application. Despite of choices made by the vendor to implement the services described in the specification (thanks to its design), enterprise beans are portable from one execution environment to another. The quality of service required by them is described in a deployment descriptor, which is analyzed at deployment time. This allows flexibility and component reuse.

The architecture of Enterprise JavaBeans allows reuse of server-side components in applications. Containers are managed by Enterprise JavaServer. A simple model of this architecture is shown on Figure 2.

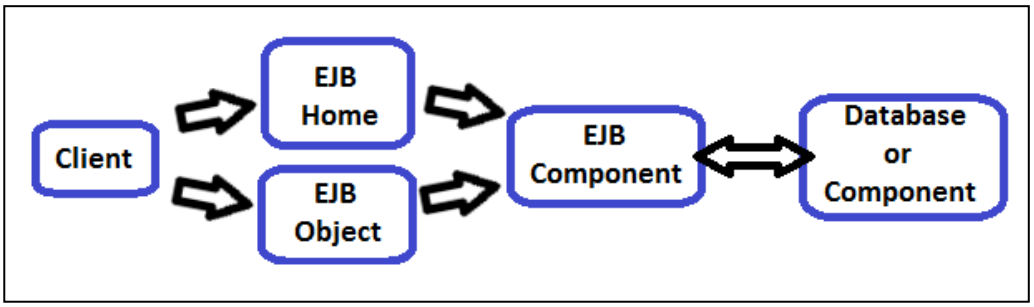

FIGURE 2: Enterprise Java Beans Architecture

Services for enterprise bean instances are provided by EJB containers – they create bean instances, manage pools and destruction of those. Besides that, containers allow services such as security and persistency to the beans they manage. Services of a container are used to call out an enterprise bean and its service level is specified at deployment time. The container must communicate with the enterprise server (with services implementation) to get access to the service of the enterprise-bean instance – transaction and naming services of JNDI (Java Naming Directory Interface) must be provided by EJS (e.g. WebSphere Application Server). It is required that for each

application component type in Java 2 Enterprise Edition there must be one container. Thanks to the existence of container between components of the application and the set of services a view of the APIs can be provided. Different containers provide different services:

- Application client container – supports components of client application
- Applet container – supports the applet programming model
- Web container – is responsible for processing requests for servlets, JSP files, coding.
- EJB container delivers interface between the application server and enterprise beans that are deployed.

To handle bean instances, a factory service is used – delivered thanks to EJBHome class, it implements home interface definition used by client programs methods. There is one EJBHome for each enterprise bean type. Before creating and finding instances, a client must locate the EJBHome class, so it can be used. Remote interface (that defines business methods of the enterprise bean) is implemented by the EJBObject class – at deployment time, which class is created from this interface definition by the container. Enterprise bean instance is created when the client program gets access to the home object and after request is sent to the container. The implementation of methods defined in the specification and those defined on the remote interface are contained within the enterprise bean class and they will be invoked by container when needed.

Data is represented by entity beans - usually it is one row of table in a database. Analogously, creating an entity bean is like adding a record to the table. If a bean instance is removed, it can be created again (using its persistent state). There can be multiple users sharing the same entity beans.
Persistency of entity beans can be handled by:

- The developer – it is bean managed persistence (BMP); the developer is responsible for providing code for the container to use (when it needs to save or restore the state of the enterprise bean)
- The container – it is container managed persistence (CMP); in this type, all persistency is delegated to the container.

Session instance bean belongs to a client. Its task is to maintain all the information as long as a client is connected (which all are lost when e.g. a server crashes or a client leaves). As/For example, a shopping cart can be used. As long as clients are connected, all their chosen products must be stored, however, when they leave, everything is deleted. Usually, a container must serve multiple clients, so it creates a pool of session bean instances. As was shown on Figure 3, there are two types of session beans:

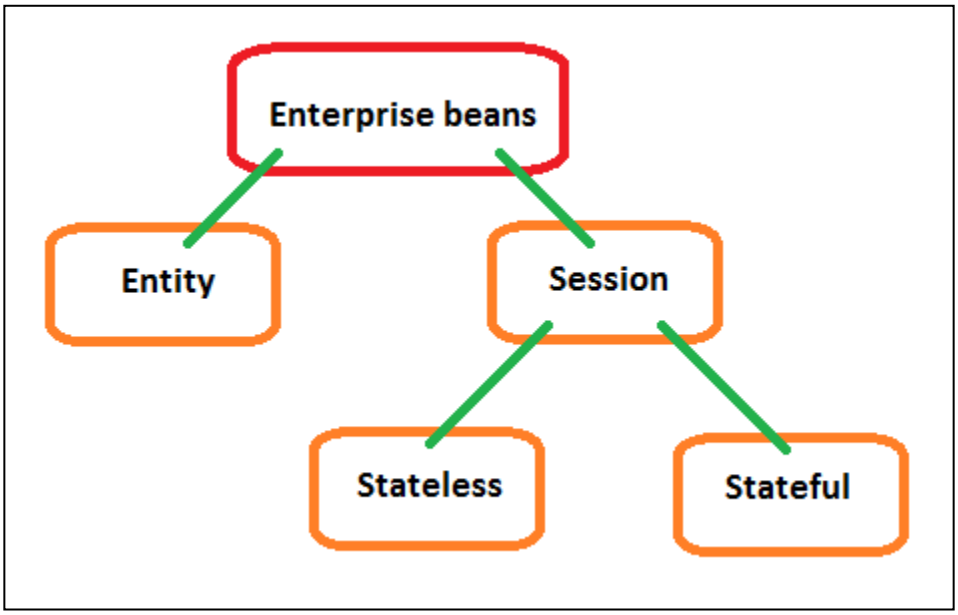

FIGURE 3: Division of enterprise beans

- Stateful – the same bean instance must be allocated by the container for every method call. New instances are created? when (on the home interface) create() is called out by the client
- Stateless – different bean instance can be allocated by the container for every method call, which means two various methods invokes can be served by a different bean instance. These session beans are effective, because the container does not need a great number of instances in order to serve a large number of clients.

3 IMPLEMENTATION

3.1 Setting environment

To start the development of an application written with the aid of Groovy, there is necessity to prepare an appropriate environment. There are few common possibilities:

- Groovy console coming within installation of Groovy
- Eclipse plugin
- IntelliJ IDEA plugin.

Using basic console is good to get to know language and its differences compare to Java, but it could be difficult to implement any advanced solutions. In addition, console does not feature text correction, hints about methods, imports and any advanced possibilities provided by other IDE. Eclipse and IntelliJ IDEA IDEs allow integrated plugins, which work as the console, but have additional helpful features. During implementation, following technologies will be used:

- IntelliJ IDEA (version 11.0.2)
- Groovy (version 1.8.6)

- Java Development Kit (JDK, version 1.7.0_03)
- Additional libraries (demonstrated later)
- Madisons default shop (provided by IBM).

The installation of first three listed above was made with the default settings of Windows installers, thus it is not shown here. To work on IntelliJ IDEA with Groovy projects it is required to go to the File -> Settings -> Plugins, choose Groovy and install it. During creation of a new project, JSDK and Groovy SDK must be chosen – it is important to do it correctly, since otherwise there is no possibility to correctly compile and run the applications. These settings can be changed later by modifying done in File -> Project Structure. Default paths to those (after installation with default settings, which was mentioned earlier) are shown on Figure 4 and Figure 5:

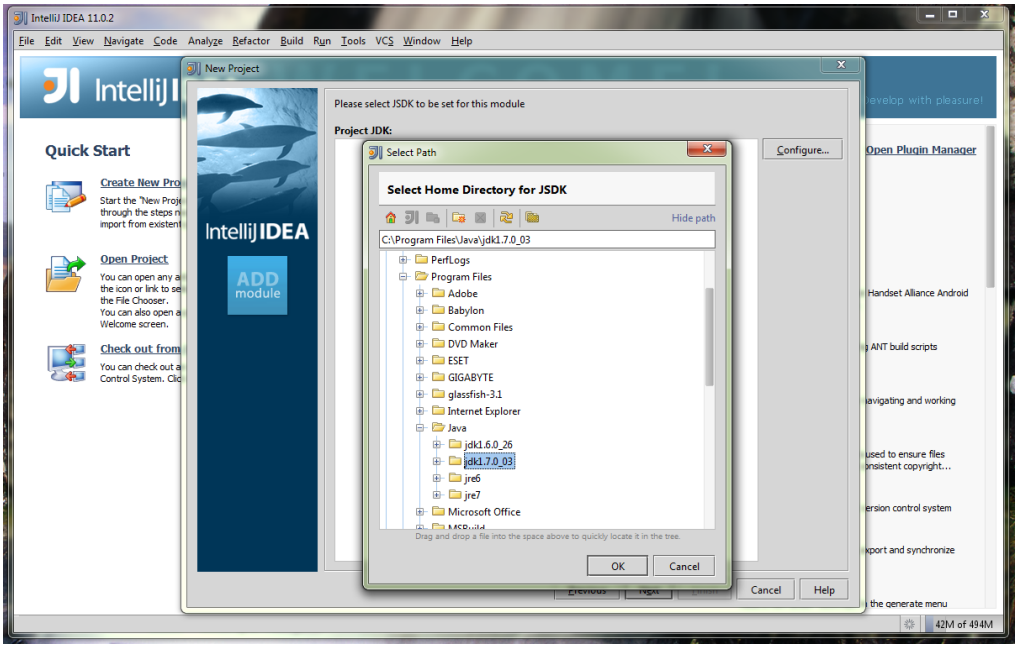

FIGURE 4: JSDK selection

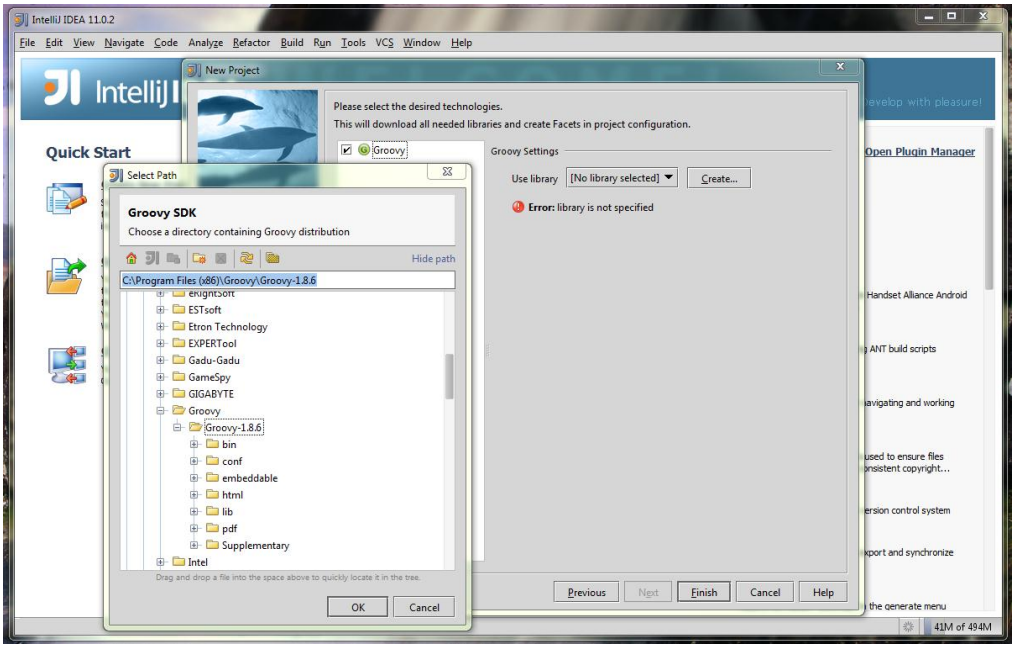

FIGURE 5: Groovy default SDK path selection

After that, the environment is set and ready for some basics applications, like Hello World. For enabling a possibility of creating more advanced applications, additional libraries must be installed and added in File -> Project Structure -> Modules-> Dependencies listed below are as follows:

- Database connection:
  - ✓ db2jcc.jar
  - ✓ db2jcc_license_cu.jar

- Charts handling:
  - ✓ jfreechart-1.0.14.jar
  - ✓ jcommon-1.0.17.jar

- Excel (.xls / .xlsx) files handling:
  - ✓ poi-3.8-beta5-20111217.jar
  - ✓ poi-excelant-3.8-beta5-20111217.jar

- ✓ poi-ooxml-3.8-beta5-20111217.jar
- ✓ poi-oxml-schemas-3.8-beta5-20111217.jar
- ✓ poi-scratchpad-3.8-beta5-20111217.jar.

3.2 Basic database connection

To have a possibility of work with business data, it is required to have a connection to database. Groovy implements methods allowing free access to it within its library groovy.sql.Sql. The line shown below is responsible for establishing the connection and assigning it to a variable (for other commands to be used):

```
def connection = Sql.newInstance("database_url",
                                 "database_login",
                                 "database_password",
                                 "database_driver")
```

During the connection test process an error occurred:

```
java.lang.ClassNotFoundException: driver
```

After searching, the solution was found - it was necessary to import two libraries required for correct connection with the database - see 3.1 Setting environment (Fadel, A. 2011. DB2 java.lang.ClassNotFoundException: com.ibm.db2.jccDB2Driver. http://ahmedfadelblog.blogspot.com/2011/06/db2-javalangclassnotfoundexception.html). When the connection was established, it was possible to start working with data such as retrieving it from database:

```
connection.eachRow("select catentry_id from catentry where
startdate = '2012-03-14-00.00.00.000000'")
{println "${it.catentry_id}"}
```

As written in documentation of groovy.sql.Sql, this call of eachRow method performs the given SQL query, calling the given Clossure with each row of the result set – basically in this case, every retrieved row (id of product, which was entered into database on 01.03 current year) is displayed in the console as shown on Figure 6:

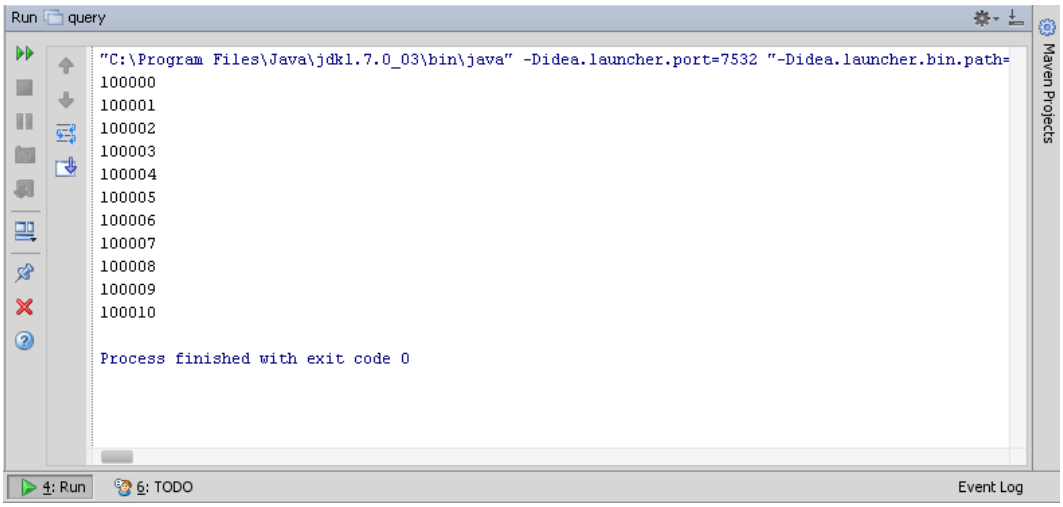

FIGURE 6: Simple query console output

There is also possibility to enhance the way queries are executed, by using additional feature provided by Groovy, which is DataSet - it is a class that provides help in establishing connection with database and processing queries, using for this purpose operators and POGO fields (Stevenson, C., King, P., Strachan, J. DataSet class documentation. Accessed on 15 March 2012. http://groovy.codehaus.org/api/groovy/sql/DataSet.html)

3.3 Advanced database connection

The example shown in section 3.1 is straightforward and its aim is to show how easily data can be obtained and presented. However, it is not a good idea to maintain connection establishment, query and other methods (if there are any) in one file, due to security and confidentiality. To improve those it can be divided into separate files, which would also make parts of script easier for implementing in other scripts. That kind of solution does not require advanced implementation and can be easily conducted. Starting from this paragraph, divide shown on Figure 7 will be used as basic configuration thus later it will not be discussed anymore.

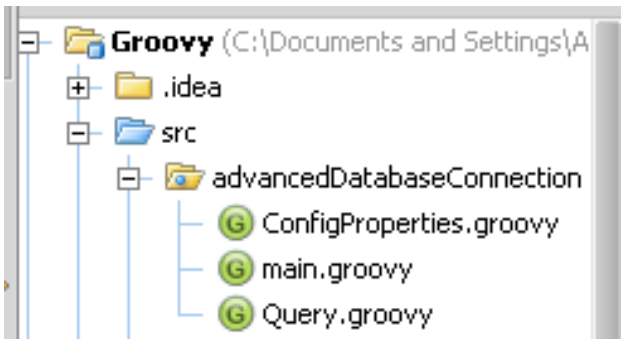

FIGURE 7: Scripts structure

As the naming suggests, in ConfigProperties.groovy all configuration data required for establishing database connection is stored there, which is presented on Figure 8:

FIGURE 8: Database configuration settings

Query.groovy contains (in this and next examples) every query, which was used to retrieve data from database with methods responsible for access to those queries. In main.groovy, connection with db2 is established. Variable *environment* enables the possibility of switching between "workspaces", e.g. from production server to localhost (without necessity of changing URL, login and others every time). Next, GroovyClassLoader and ConfigSlurper are used to load configuration from ConfigProperties:

- GroovyClassLoader is a ClassLoader, which can load GroovyClass. Classes that are loaded are also cached. GroovyClassLoader is used to fix ClassNotFound exception (Strachan, J., Laforge, G., Goetze, S., Ran, B., Stirling, S., Theodorou, J. GroovyClassLoader class documentation. Accessed on 16 March 2012. http://groovy.codehaus.org/api/groovy/lang/GroovyClassLoader.html)
- ConfigSlurper is a utility class, which purpose is to write properties file (e.g. scripts) to execute configuration (Rocher, G. ConfigSlurper class documentation. Accessed on 16 March 2012. http://groovy.codehaus.org/gapi/groovy/util/ConfigSlurper.html)

Establishing connection with database by using GroovyClassLoader and ConfigSlurper is shown on Figure 9:

```
def environment = "prod"

GroovyClassLoader classLoader = new GroovyClassLoader(getClass()
                                    .getClassLoader())
def config = new ConfigSlurper(environment)
                .parse(classLoader
                .loadClass('ConfigProperties'))

connection = Sql.newInstance(config.db.url,
                             config.db.username,
                             config.db.password,
                             config.db.driver)
```

FIGURE 9: Setting up database connection

Finally, a new connection is established and queries can be called out by using variable *connection* and the results are printed in console. Figure 10 presents how queries are executed on the Query object *qu*:

```
Query qu = new Query()
    connection.eachRow(qu.firstQuery())
            {println "${it.FirstQueryResult}"}
    connection.eachRow(qu.secondQuery())
            {println "${it.SecondQueryResult}"}
```

FIGURE 10: Queries call out

That kind of division has many benefits: the code is cleaner, the connection and queries are gathered into separated places, thus finding and modifying them is easier.

Example queries that can be useful within this application:

- Retrieving promotion codes associated to promotion:

```
Select code from px_promotion where px_promotion_id =
known_promotion_id
```

- Checking if item has inventory:

```
Select catentry_id, quantity from inventory where catentry_id
= known_product_id
```

3.4 Sending data to HTML file

Previous examples are very useful for users who know basic SQL alongside with the meaning of different fields of database, because all returned data were printed in basic console output. However, when a person is inexperienced with those or has no need of knowing anything about technical details, the situation requires other solutions. Exporting all retrieved data to HTML file is one of the possibilities. Connection to database and handling queries does not change (compering to the last example). By following instructions on generating HTML report (Bashar, A.-J., Groovy and Grails Recipes, 2009, 146), it is required to add a fragment responsible for handling HTML part to the last example. For this purpose, two additional features will be used:

- FileWriter is a class which purpose is to write characters to a file (to be precise: to write characters streams); if writing streams of raw bytes is required, it is better to use FileOutputStream (FileWriter class documentation. Accessed on 18 March 2012. http://docs.oracle.com/javase/1.4.2/docs/api/java/io/FileWriter.html)
- MarkupBuilder is class created for helping in the process of creation XML / HTML markup(Strachan, J., Aust, S. M., Stirling, S., King, P. MarkupBuilder class documentation. Accesed on 18 March 2012. http://groovy.codehaus.org/api/groovy/xml/MarkupBuilder.html).

First, FileWriter variable holding path and name for output file is required so it can be passed to MarkupBuilder. When this is done, the task limits to HTML knowledge and proper setting of each attribute as described below:

- Head (with webpage title)
- Body, containing main objects:
  - ✓ Table (and its title)
  - ✓ Column titles

- ✓ Content of every column.

Configuration of HTML structure that was just mentioned is shown on Figure 11:

```
Query qu = new Query()
def writer = new FileWriter('C:\\temporary\\ibm_fields.html')
def html = new groovy.xml.MarkupBuilder(writer)
html.html
        {
            head
            {
                title 'Simple html table generate'
            }
            body
                    {
                        hl 'IBM fields'
                        table(border:1)
                        {
                            th('state')
                            th('optcounter')
                            connection.eachRow( qu.firstQuery() )
                                    {
                                        field ->
                                        tr
                                        {
                                            td(field.state)
                                            td(field.optcounter)
                                        }
                                    }
                        }
                    }
        }
```

FIGURE 11: HTML file generating

To enhance final appearance of generated table, normal HTML coding style can be used, like:

- Borders
- Margins
- Background color of specific elements

- Alignment
- Nested tables
- Grouping.

When this is executed, file ‘ibm_fields.html’ is created in location given to FileWriter – it contains the described table; a part of it is presented on Figure 12:

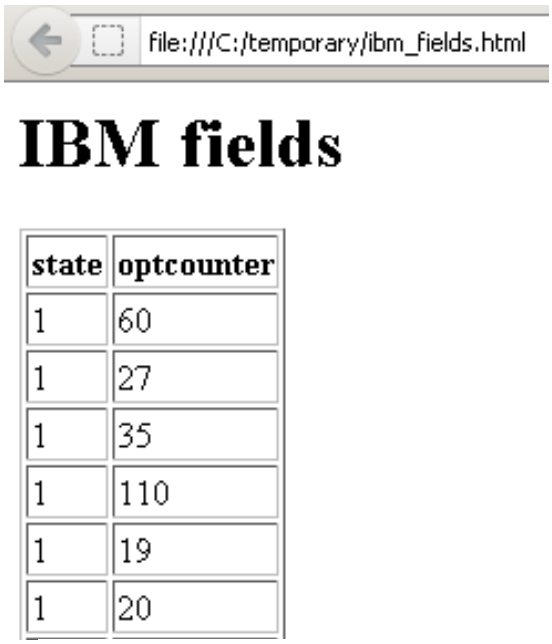

FIGURE 12: Generated table output

Data presented in that way can be shown to client or investor. By proper descriptions of fields, this is clear information to a person who receives it. An additional benefit is the mobility - .html file can be easily sent via mail or chat.

Example queries that can be useful within this application:

- All examples presented in paragraph 3.3
- Checking all products with given status:

```
select orders_id, trading_id from orderitems
where orders_id in (select orders_id from orders
where status = 'known_status')
```

- Finding all products that are not IteamBeans:

```
select orders_id from catentry where catenttype_id <>
'ItemBeans'
```

3.5 Data chart generator

Although last example showed that exporting to .html file is user-friendlier than raw console output, it still is not in form for business meetings or negotiations. Data obtained from database can also be used for creating presentations and showing statistics. Information retrieved can be typed manually into more advanced programs (MATLAB or Excel) which are able to create charts or have other methods for processing those, however it requires time and at least one person attention each time data would have to be typed. Groovy alongside with Java methods gives possibility to make whole process semi-automatic.

One of the available form to express retrieved data is to present it on chart – alongside with labels and descriptions it is a good way to explain what data represents. There are many possibilities for creating charts by using groovy – library JFreeChart can be used for basic or more advanced ones, also Google Chart API can be used for this purpose or (if using Grails) Google Chart plugin is available.

During this part of implementation, additional features were used:

- ChartUtilities is a collection of methods allowing usage of additional utilities of JFreeChart library (ChartUtilities class documentation. Accessed on 21 March 2012. www.jfree.org/jfreechart/api/javadoc/org/jfree/chart/ChartUtilities.html)

- ChartUtilities.saveChartAsPNG – depending on list of argument passed to this method, different ways for saving chart as image can be called out (there is also method saveChartAsJPEG, which allows the same executes, only the output will be .jpeg file)

3.5.1 Pie chart

For creating a pie chart a different approach was taken than when creating a line chart. Instructions for this purpose were found on webpage from reference, therefore methods and solutions shown there are not described here in detail (Vishal, S. 2010. Generating Dynamic charts in Grails. Accessed on 22 March 2012. http://www.intelligrape.com/blog/2010/03/31/generating-dyanmic-charts-in-grails/).

First of all two libraries, which were already mentioned, had to be imported (jfreechart-1.0.10.jar and jcommon-1.0.13.jar). When it was done, it was necessary to create the whole catalogue structure, basing on the fundaments created in paragraph 3.3.

That catalogue structure is shown on Figure 13:

FIGURE 13: Catalog structure for chart application

The main issue after implementing methods responsible for chart and label generation was to export and convert the data retrieved by queries, therefore the method drawPieChart would be able to use it for its purpose. One of possible solution is to modify the way of handling query results already shown:

```
connection.eachRow(qu.firstQuery())
{println "${it.firstQueryResult}"}
```

Into call out which will allow holding and reusing the retrieved data – in place of println method it is enough to put assigning to a variable, like in the example below:

```
def result1
connection.eachRow(qu.firstQuery())
{result1 = "${it.firstQueryResult}"}
```

Arguments of drawPieChart are:

```
void drawPieChart(List keys, List values, Integer width,
Integer height)
```

Where width and height are fixed sizes of a chart, keys are labels for descriptions and values are data, which creates the chart. It is not visible, but List values must contain only Integer elements – for this reason, variable result1 must be cast from its current type, to Integer (variable could not be declared as Integer, because later it is used as container for String). In Groovy it can be done as shown on the code snippet presented below:

```
int result1_toInt
result1_toInt = result1.toInteger()
```

When all results were cast into Integer, the only issue that was left to be done was to create two lists containing labels and data. After that, all acquired values had to be inserted into them – so that drawPieChart could be called out with all its arguments (without explicit cast, compiler returns an error stating about missing method call out). The snippet presented below shows how it can look in practice (values typed as width and height are typed manually, but those can also be asked from user):

```
def labelList = []
def valueList = []
labelList = [label1, label2, label3, label4, label5, labe6]
valueList = [result1_toInt, result2_toInt, result3_toInt,
             result4_toInt, result5_toint, result6_toInt]
drawPieChart( labelList, valueList, 700, 500)
```

Depending on the complexity of the called out queries obtaining result could take some time. After it is finished, on the default location, a .png is created, which is presented on FIGURE 14:

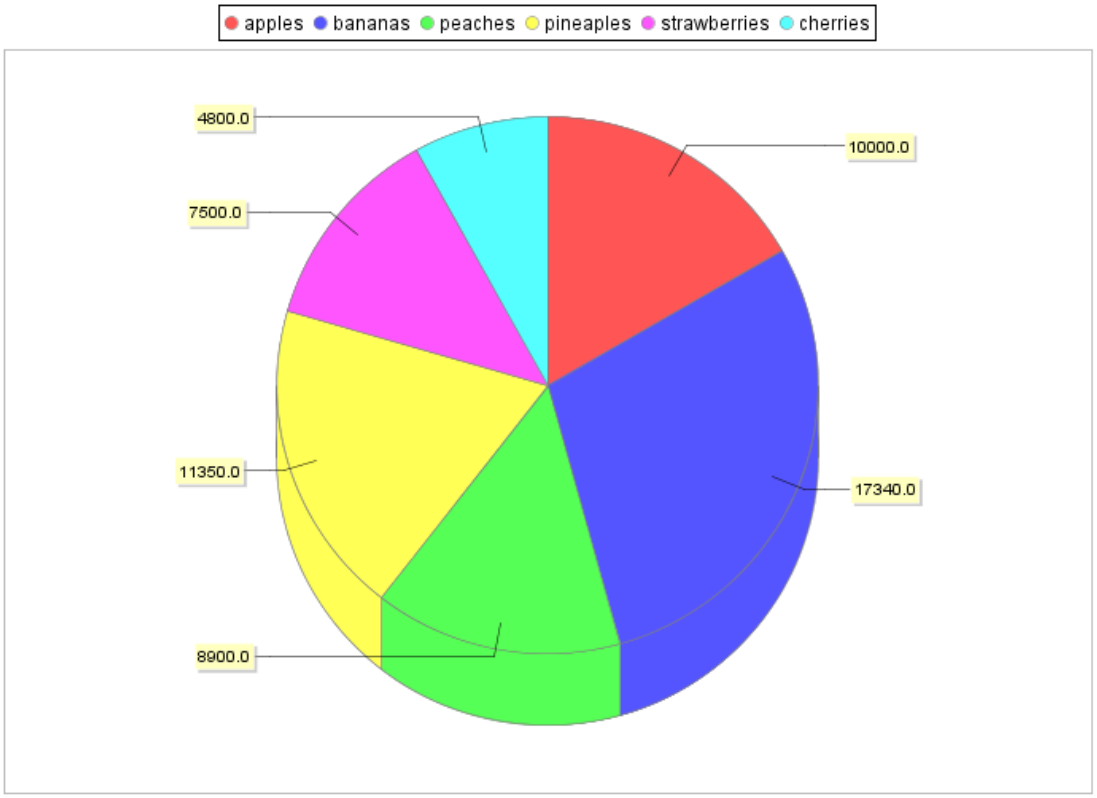

FIGURE 14: Pie chart example

Example queries that can be useful within this application:

- Preparing statistics of what status orders have:

```
select count(orders_id) as result1 from orderitems where
orders_id in (
select orders_id from orders where status = 'known_status')

select count(orders_id) as result2 from orderitems where
orders_id in (
select orders_id from orders where status =
'different_known_status')
```

etc., until all status are checked

- Preparing statistics of registered customers' gender divide:

```
select count(users_id) as result1 from userdemo
where gender = 'F' and users_id in (
select users_id from users where registertype = 'R')

select count(users_id) as result2 from userdemo
where gender = 'M and users_id in (
select users_id from users where registertype = 'R')

select count(users_id) as result3 from userdemo
where gender = 'N' and users_id in (
select users_id from users where registertype = 'R')
```

Where F – female, M- male, N – not provided (default, if others not provided)

3.5.2 Line chart

To create complete line chart with using Groovy, sample from reference was taken as starting point (Accessed on 22 March 2012. http://groovy.codehaus.org/Plotting+graphs+with+JFreeChart). This example creates line chart with static data using for that purpose Swing library. In this implementation, the following elements were exchanged:

- Static data input was replaced with converted results of queries (result1_toInt, as in paragraph 3.5.1)
- Use of Swing was not necessary, so it was removed.
- Exporting chart to image format was required.

When chart was generated it had to be saved as file. It was possible to achieve thanks to use of method presented on the snippet below (image does not have to be saved as PNG with only those results – method *saveChartAsPNG* has also other call outs; alternatively *saveChartAsJPEG* can be used):

```
def writer = new File('C:\\temporary\\linechart.png')
ChartUtilities.saveChartAsPNG(writer, chart, 700, 500)
```

The result of this program is shown on FIGURE 15:

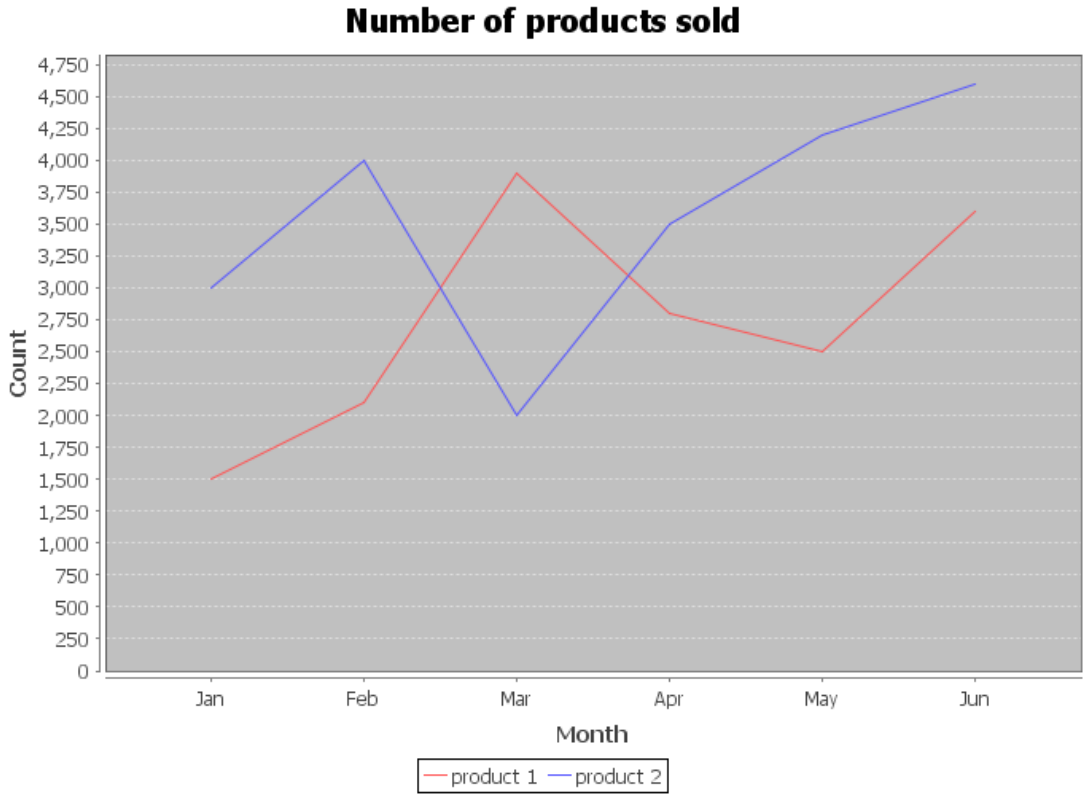

FIGURE 15: Line chart example

Example query that can be useful within this application:

Preparing statistics of users' registration dates:

```
Select count(users_id) as result1 from users
where registration = 'known_date'

Select count(users_id) as result1 from users
where registration = 'other_known_date'
```

etc., until conditions given by client are fulfilled.

3.6 URLs parsing

Creating a product catalogue requires the latest information – the folder must be accurate and consistent with content, which is available for customer to see and obtain using the application. For that purpose, products have two unique numbers: their ID and barcode, both stored in database. When the catalogue was to be created, there was a request for confirming cohesion between what a client had and what was stored in database. For this purpose, parsing URLs (basing on .xls / .xlsx files) was the main execution method.

The first important issue was to correctly handle .xls / .xlsx file. After connection with the database was established (for this purpose, the example from the previous paragraphs was used), there was a necessity to open Microsoft Office Excel file, to obtain data that would serve as the source for what the queries should concern. For this purpose, additional features had to be used:

- FileInputStream is a class the purpose of which is to get input bytes from a file – it is meant for reading raw bytes streams (Accessed on 23 March 2012. http://docs.oracle.com/javase/1.4.2/docs/api/java/io/FileInputStream.html)
- Apache library for handling Excel files.

The usage of the aforementioned modules is presented on the snippet bellow:

```
FileInputStream fis = new FileInputStream(inFile)
org.apache.poi.ss.usermodel.Workbook wb =
org.apache.poi.ss.usermodel.WorkbookFactory.create(fis);
org.apache.poi.ss.usermodel.Sheet sheet =
wb.getSheetAt(sheetIdx);

for (org.apache.poi.ss.usermodel.Row row : sheet) {
      parseXLSRow(row)
}
```

At this point, .xls / .xlsx file was open; however, there was no possibility to work with data provided that way (without additional methods). Changing this fact was possible by converting Microsoft Office Excel basic structures (cells) into a format that can be handled by Groovy (e.g. Integer or String). That kind of parsing was achieved by writing method cellToString – it was responsible for creating a new (null) cell and assigning its type as String, copying the content of an important cell into that new cell.

When it was done, all unwanted symbols were cast out and the content could be returned as basic String, which is shown on Figure 16:

```
String cellToString(org.apache.poi.ss.usermodel.Row row, int cellNumber){
        String str = null;
        org.apache.poi.ss.usermodel.Cell cell
        cell = row.getCell(cellNumber, org.apache.poi.ss.usermodel.Row.CREATE_NULL_AS_BLANK)
        cell.setCellType(cell.CELL_TYPE_STRING)
        str = cell.toString()
        if (str.contains("\"") || str.contains("*")){
                str = str.replaceAll(/\"/,"\"\"")
        }
        str = "\""+str+"\""
        str = str.replaceAll("\\u00A0|\\n|\\r", "") //Replace the odd, non breaking space ASCII
        return str
}
```

FIGURE 16: Function parsing Excel cells to Strings

Next, that returned String had to be checked and (eventually) corrected – the proper length, symbols and confirmation, so that nothing was lost in Microsoft Office Excel cells due to the format or conversion. Assuming that String fulfilled all requirements, it was sent to be part of queries – retrieving product ID depended on it. When ID was obtained, there was a way of getting all the remaining details, like name (alongside with normalizing it) and the whole URL address (also, with checking that the category structure was preserved). The output can be adjusted, depending on clients' needs. It can be a raw console output pasted into an email or the links can be directly saved into text file, which can be sent as an attachment.

4 RESULTS

The implementation shows that the data obtained from database can be handled in many ways, where the condition was: what should be the final result? Smoothing the console output was shown by implementing data processing into:

- .html file
- Charts, depending on the situation
  - ✓ Pie (PiePlot3D; .png file created)
  - ✓ Line (ChartFactory; Swing packet output or .png file)

Besides that, the last example in the implementation (paragraph 3.6 URLs parsing) proves, how great help Groovy provides: in a few minutes, all products which were required for catalogue creation were obtained based on their barcodes, whereas without using the parsing script it could take few hours to check those manually.

The benefits, which come alongside with using Groovy, are numerous, which was proven during the implementation process and can be shown in a form of list:

- Groovy is not a new language to learn, basic knowledge of Java is required to develop applications based on it
- Groovy is simpler and easier in use than Java, yet it enables additional methods and other features which do not exist in Java
- Groovy enables multiple ways for data processing and presentation of results

As a proof, a comparison between basic Java and Groovy Java Beans size and complexity can be done, as shown on Figure 17 and Figure 18:

```java
import java.math.BigDecimal;

public class Product {
   private String brand;
   private String name;
   private int barcode;
   private String description;
   private BigDecimal price;

   public Product(String brand, String name, int barcode, String description) {
      this.brand = brand;
      this.name = name;
      this.barcode = barcode;
      this.description = description;
   }

   public String toString() {
      return "Product : " +
            "brand -- " + brand +
            "\n: name -- " + name +
            "\n: barcode -- " + barcode +
            "\n: description -- " + description +
            ".\n"; }

   public String getBrand() { return brand; }

   public void setBrand(String brand) {
      this.brand = brand;
   }

   public String getName() { return name; }

   public void setName(String name) { this.name = name; }

   public int getBarcode() { return barcode; }

   public void setBarcode(int barcode) { this.barcode = barcode; }

   public String getDescription() { return description; }

   public void setDescription(String description) { this.description = description; }

   public BigDecimal getPrice() { return price; }

   public void setPrice(BigDecimal price) {
     this.price = price.setScale(3, BigDecimal.ROUND_HALF_UP);
   }
}
```

FIGURE 17: JavaBean written in Java

```
class Product {
    String brand
    String name
    Integer barcode
    String description
    BigDecimal price

    public void setPrice(BigDecimal newPrice){
        price = newPrice.setScale(3, BigDecimal.ROUND_HALF_UP)
    }

    public String toString() {
        return """Product:
        brand -- ${brand}
        name -- ${name}
        barcode -- ${barcode}
        """
    }
}
```

FIGURE 18: JavaBean written in Groovy

For confirming also that the data was retrieved as a basic console output, an additional example can be presented – such as obtaining promotion codes of specific promotion from database. To fulfill this purpose, the following query was used:

```
select code from px_cdpool where px_cdpool_id in(
select px_cdpool_id from px_cdpromo
where px_promotion_id in (
select px_promotion_id from px_promotion
where storeent_id = 10051
order by lastupdate desc fetch first 1 row only))"
```

Using the presented query for this purpose in the application shown in paragraph 3.3, the information retrieved looks like the one presented on Figure 19:

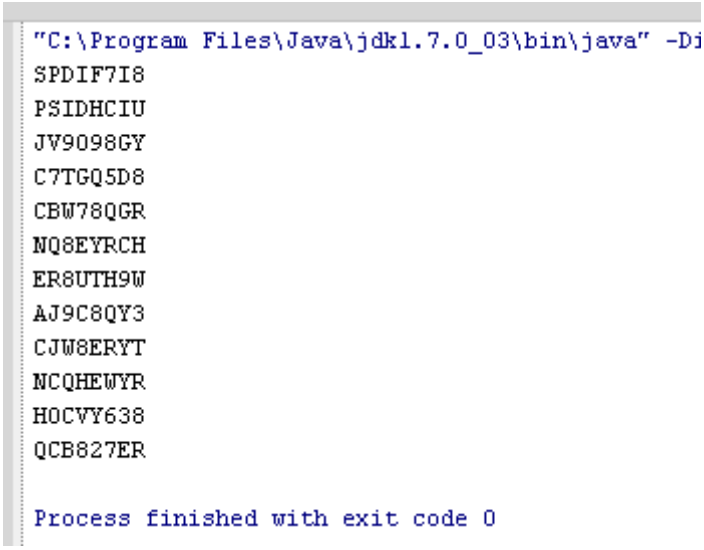

FIGURE 19: Acquiring promotion codes - console output

To show the contrast in output, this query was also used also in the application presented in paragraph 3.4 (with a modification of HTML file) – the output of this is presented on Figure 20:

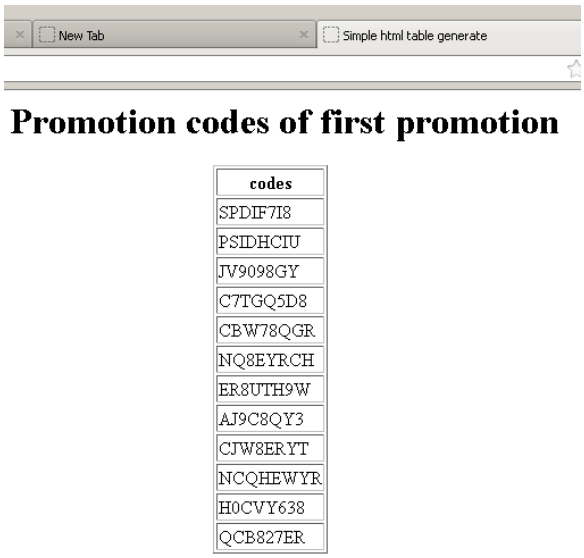

FIGURE 20: Acquiring promotion codes - HTML file output

The difference in those two outputs is obvious and undeniably shows the advantage of .html file. When it adds with other benefits that were listed, it makes the script destined for processing data into webpage a useful tool.

The most significant dissimilarity can be observed between the console output and the charts. Despite the fact that the second one is better when a quick analysis of data is needed and use information is provided in the work (e.g. fixing missing products or analyzing why status of order stopped on 'pending' and is not processing to 'sent to the customer') in situations that require attention and understating of a third person (e.g. client), charts are better for presenting the gathered information. To visualize it, the queries below (showing how much products from the database could be bought) were used in applications from paragraphs 3.3 and 3.5.1:

```
select count(catentry_id) from catentry
where buyable = '1'"

select count(catentry_id) from catentry
where buyable = '0"
```

Figure 21 shows the output from the console:

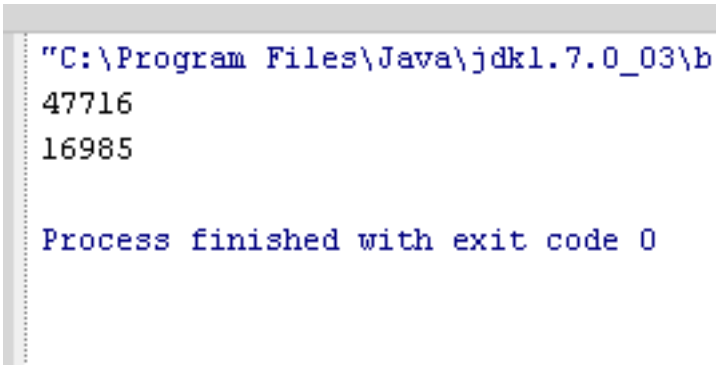

FIGURE 21: Checking product status – console output

A result presented in that way is totally useless for someone not familiar with database structure. The output shown in the Figure 22 presents results of execution of the second mentioned application, which is more convenient and user-friendly:

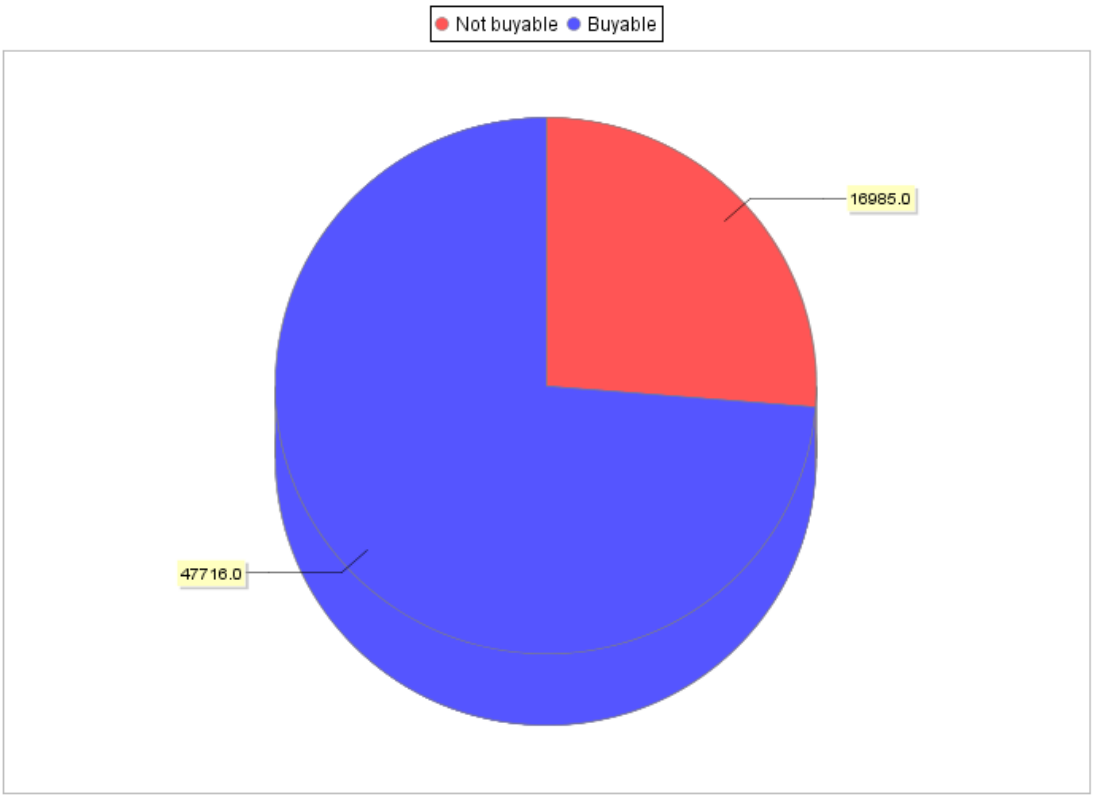

FIGURE 22: Checking product status – PNG file output

There are many ways to improve the applications presented in the implementation:

- During the whole research, data (like queries, labels) was input manually by the programmer, which normally is forbidden for a simple reason: the client usually does not have access to that part of the code, therefore everything should be available to be input dynamically. It is possible to write additional scripts that would handle the data input, by dialog windows or by asking for input file with all required Strings
- Charts presented in paragraph 3.5 (despite the fact that they present different data gathered) were all made by using JFreeChart library. It is not

the only possible way to handle charts in Groovy – as it was mentioned, Google Chart API and Google chart plugin also give that possibility.

5 DISCUSSION

The main idea of the thesis was to show a dialog between database and client, which has to be served by a third person from a support team – that implicates the way of data input: manual, not using additional scripts. Implementing also that part of the problem would make the code too complex and could give the reader a false view of the objective of this research.

Initially, there was an idea to show also a different approach in retrieving data from a database, namely by using (include in groovy) Dataset, however, this idea was dropped in the process of the implementation. Despite the fact that it is relatively simple to understand and implement it would have made unnecessary complications. Dataset use is good in applications designed for real alongside WebSphere commerce applications; as far as research that is focused on showing an idea is concerned, it would be a case of 'form over content'.

During the research process there was a great deal of new issues. The first and the most important one was Groovy language itself. Helpful with getting familiar with it were the facts that it was based on Java language with only additional features. There were many clues and hints for beginners listed in the sources below:

- http://groovy.codehaus.org/Getting+Startd+Guide
- Beginning Groovy and Grails by Judd Christopher M.
- Groovy and Grails Recipes by Bashar A.-J

It is quite simple to start developing with the aid of Groovy, thanks to the hypertrophied community and easy access to sources. Documentation on

http://groovy.codehaus.org and http://docs.oracle.com helps with finding an answer for almost every basic issue; there are also many forums and blogs (like the one presented in paragraph 3.2) which have accurate information about fixing specific errors.

Groovy is not designed only for this concrete platform (WebSphere commerce) - it is a multiplatform and can be implemented in many environments. Despite that, it is not a great trouble to make it work properly on that kind of highly specialized platform. It would not have taken a considerable amount of time to prepare a whole environment working (for basic applications) and a little longer to make more advanced program functioning.

Knowing basics was fundamental for solving later issues, namely such as the one described in paragraph 3.2 or Caught:groovy.lang.MissingMethodException: No signature of method (which can occur if wrong arguments are passed).

After getting familiar with the basics of Groovy, it was possible to start developing all planned applications. During the implementation, different approaches were chosen to present possibilities of data processing and ways of showing it to the third person. Creating .html file enables mobility of data and also remote access to it without interference with the input. The same benefits come from creating charts; additionally in those cases data is available in the form proper for presentations. In addition, information shown that way is not too complex and is easier to understand. URL parsing shown in the last paragraph of the implementation serves as a practical example that could be ‘as is’ implemented into working WebSphere commerce application, without the necessity of great load of code refactoring.

Charts were created only by the use of JFreeChart library to make the complexity of code as simple as possible. It was not the only possible way for drawing those (which also had been mentioned), however, it was the only manner shown and it was done on purpose – again it was for the sake of the main objective of this thesis.

REFERENCES

Bashar, A.-J. 2009. Groovy and Grails Recipes. New York: Apress

Endrei, M., Cluning, R., Daomanee, W., Heyward, J., Iyengar, A., Mauny, I., Naumann, T., Sanchez, A. 2002. IBM WebSphere V4.0 Advanced Edition Handbook.

Fadel, A. 2011. DB2 java.lang.ClassNotFoundException: com.ibm.db2.jccDB2Driver. http://ahmedfadelblog.blogspot.com/2011/06/db2-javalangclassnotfoundexception.html

Judd, C. M., Nusairat, J. F., Shingler, J. 2009. Beginning Groovy and Grails: From Novice to Professional. New York: Apress.

Koenig, D., Glover, A., King, P., Laforge, G. & Skeet, J. 2007. Groovy in Action. New York: Manning.

Vishal, S. 2010. Generating Dynamic charts in Grails. http://www.intelligrape.com/blog/2010/03/31/generating-dyanmic-charts-in-grails/

http://docs.oracle.com/

http://groovy.codehaus.org/

www.jfree.org/jfreechart/api/javadoc/org/jfree/chart/ChartUtilities.html

APPENDICES

APPENDIX 1. WebSphere Commerce database table: CATENTRY

This table holds the information related to a CatalogEntry. Examples of CatalogEntries include Products, Items, Packages, and Bundles.

Column Name	Column Type	Column Description
CATENTRY_ID	BIGINT NOT NULL	The internal reference number of the CatalogEntry.
MEMBER_ID	BIGINT NOT NULL	The reference number that identifies the owner of the CatalogEntry. Along with the PARTNUMBER, these columns are a unique index.
ITEMSPC_ID	BIGINT NULL	The SpecifiedItem that this CatalogEntry relates to. This column should only be populated for CatalogEntries that are of type "ItemBean", "PackageBean", or "DynamicKitBean". SpecifiedItems are used for fulfillment.
CATENTTYPE_ID	CHARACTER (16) NOT NULL	Identifies the type of CatalogEntry. Foreign key to the CATENTTYPE table. The supported default types are: ProductBean, ItemBean, PackageBean, BundleBean and DynamicKitBean.
PARTNUMBER	VARCHAR (64) NOT NULL	The reference number that identifies the part number of the CatalogEntry. Along with the MEMBER_ID, these columns are a unique index.
MFPARTNUMBER	VARCHAR (64) NOT NULL	The part number used by the manufacturer to identify this CatalogEntry.
MFNAME	VARCHAR (64) NOT NULL	The name of the manufacturer of this CatalogEntry.
MARKFORDELETE	INTEGER NOT NULL	Indicates if this CatalogEntry has been marked for deletion: 0 = No. 1 = Yes.
URL	VARCHAR (254) NULL	The URL to this CatalogEntry, which can be used as a download URL for soft goods.
FIELD1	INTEGER	Customizable.

	NULL	
FIELD2	INTEGER NULL	Customizable.
LASTUPDATE	TIMESTAMP NULL	Indicates the last time the CatalogEntry was updated.
FIELD3	DECIMAL (20,5) NULL	Customizable.
ONSPECIAL	INTEGER NULL	This flag identifies if this CatalogEntry is on special.
ONAUCTION	INTEGER NULL	This flag identifies if this CatalogEntry is on auction.
FIELD4	VARCHAR (254) NULL	Customizable.
FIELD5	VARCHAR (254) NULL	Customizable.
BUYABLE	INTEGER NULL	Indicates whether this CatalogEntry can be purchased individually: 1=yes and 0=no.
OID	VARCHAR (64) NOT NULL	Reserved for IBM internal use.
BASEITEM_ID	BIGINT NULL	The BaseItem that this CatalogEntry relates to. This column should only be populated for CatalogEntries that are of type ProductBean, PackageBean or DynamicKitBean. BaseItems are used for fulfillment.
STATE	CHARACTER (1) NULL, DEFAULT '1'	Reserved for IBM internal use.
STARTDATE	TIMESTAMP NULL	The date when this catalog entry is introduced.
ENDDATE	TIMESTAMP NULL	The date when this catalog entry is withdrawn.
RANK	DOUBLE NULL	Relative search ranking.

APPENDIX 2. WebSphere Commerce database table: CATENTTYPE

This table defines all possible types of CatalogEntries. Examples of CatalogEntry types are ProductBean, ItemBean, PackageBean, BundleBean and DynamicKitBean.

Column Name	Column Type	Column Description
CATENTTYPE_ID	CHARACTER (16) NOT NULL	The identifier of this CatalogEntry type.
DESCRIPTION	VARCHAR (254) NULL	The description for this CatalogEntry type.
OID	VARCHAR (64) NULL	Reserved for IBM internal use.

APPENDIX 3. WebSphere Commerce database table: INVENTORY

The inventory table. Each row of this table contains a quantity amount representing the inventory for a particular CatalogEntry available to be shipped from a FulfillmentCenter on behalf of a Store. This table cannot be used in conjunction with Available To Promise (ATP) inventory allocation. It is used only when ATP inventory is not enabled (refer to the ALLOCATIONGOODFOR column of the STORE table), and is used by the default implementations of the non-ATP inventory task commands: ResolveFullmentCenterCmd, CheckInventoryCmd, UpdateInventoryCmd, and ReverseUpdateInventoryCmd.

Column Name	Column Type	Column Description
CATENTRY_ID	BIGINT,NOT NULL	The CatalogEntry.
QUANTITY	DOUBLE NOT NULL, DEFAULT 0	The quantity amount, in units indicated by QUANTITYMEASURE.
FFMCENTER_ID	INTEGER, NOT NULL	The FulfillmentCenter.
STORE_ID	INTEGER, NOT NULL	The Store.
QUANTITYMEASURE	CHARACTER (16) NOT NULL, DEFAULT 'C62'	The unit of measurement for QUANTITY.
INVENTORYFLAGS	INTEGER NOT NULL, DEFAULT 0	Bit flags, from low to high order, indicating how QUANTITY is used: 1 = noUpdate. The default UpdateInventory task command does not update QUANTITY. 2 = noCheck. The default CheckInventory and UpdateInventory task commands do not check QUANTITY

APPENDIX 4. WebSphere Commerce database table: ORDERITEMS

Each row of this table represents an OrderItem in an Order.

Column Name	Column Type	Column Description
ORDERITEMS_ID	BIGINT NOT NULL	Generated unique key.
STOREENT_ID	INTEGER NOT NULL	The StoreEntity the Order (this OrderItem is part of) is part of. This is normally a Store unless STATUS is Q, in which case it is normally a StoreGroup.
ORDERS_ID	BIGINT NOT NULL	The Order this OrderItem is part of.
TERMCOND_ID	BIGINT NULL	The TermAndCondition, if known, that determined the price for this OrderItem.
TRADING_ID	BIGINT NULL	The TradingAgreement, if known, that determines the TermAndCondition objects (including how the price is determined) that apply to this OrderItem.
ITEMSPC_ID	BIGINT NULL	The specified item to be allocated from available inventory and shipped to the customer.
CATENTRY_ID	BIGINT NULL	The CatalogEntry, if any, of the product being purchased.
PARTNUM	VARCHAR (64) NULL	The part number of the CatalogEntry (CATENTRY.PARTNUMBER) for the product.
SHIPMODE_ID	INTEGER NULL	The ShippingMode, if still known.
FFMCENTER_ID	INTEGER NULL	The FulfillmentCenter, if known, from which the product will ship.
MEMBER_ID	BIGINT NOT NULL	The customer of the OrderItem (which is the same as the customer of the Order).
ADDRESS_ID	BIGINT NULL	The shipping Address, if any, for this OrderItem.
ALLOCADDRESS_ID	BIGINT NULL	The shipping Address used when inventory for this OrderItem was allocated or backordered.
PRICE	DECIMAL (20,5) NULL	The price for the nominal quantity of the product (CATENTSHIP.NOMINALQUANTITY).
LINEITEMTYPE	CHARACTER (4) NULL	If specified, indicates the type of the OrderItem. ALT = the OrderItem represents an alternative item (may not be exactly what the customer requested).
STATUS	CHARACTER	All OrderItems for an Order hold a copy of

	(1) NOT NULL	the Order Status. See the description of the STATUS column for the ORDERS table.
OUTPUTQ_ID	BIGINT NULL	Reserved for IBM internal use.
INVENTORYSTATUS	CHARACTER (4) NOT NULL, DEFAULT 'NALC'	The allocation status of inventory for this OrderItem: NALC = inventory is neither allocated nor backordered. BO = inventory is backordered. ALLC = inventory is allocated. FUL = inventory has been released for fulfillment.
LASTCREATE	TIMESTAMP NULL	The time this OrderItem was created.
LASTUPDATE	TIMESTAMP NULL	The most recent time this OrderItem was updated. Changing inventory allocation related information does not cause this timestamp to be updated (refer to the LASTALLOCUPDATE column).
FULFILLMENTSTATUS	CHARACTER (4) NOT NULL, DEFAULT 'INT'	The fulfillment status of the OrderItem: INT = not yet released for fulfillment. OUT = released for fulfillment. SHIP = shipment confirmed.
LASTALLOCUPDATE	TIMESTAMP NULL	The most recent time inventory was checked (for unallocated OrderItems), allocated, or backordered, for this OrderItem.
OFFER_ID	BIGINT NULL	The Offer, if any, and if it still exists, from which PRICE was obtained.
TIMERELEASED	TIMESTAMP NULL	The time this OrderItem was released for fulfillment.
TIMESHIPPED	TIMESTAMP NULL	The time this OrderItem was manifested for shipment.
CURRENCY	CHARACTER (10) NULL	The currency of OrderItem monetary amounts other than BASEPRICE. This is the same as the currency of the order, ORDERS.CURRENCY. This is a currency code as per ISO 4217 standards.
COMMENTS	VARCHAR (254) NULL	Comments from the customer, such as a greeting for a gift.
TOTALPRODUCT	DECIMAL (20,5) NULL, DEFAULT 0	PRICE times QUANTITY.
QUANTITY	DOUBLE NOT NULL	The result of multiplying QUANTITY by CATENTSHIP.NOMINALQUANTITY must be a

		multiple of CATENTSHIP.QUANTITYMULTIPLE, and represents the actual quantity being purchased, in the unit of measurement specified by CATENTSHIP.QUANTITYMEASURE.
TAXAMOUNT	DECIMAL (20,5) NULL	The total sales taxes associated with this OrderItem, in the Currency specified by CURRENCY.
TOTALADJUSTMENT	DECIMAL (20,5) NULL, DEFAULT 0	The total of the monetary amounts of the OrderItemAdjustments for this OrderItem, in the Currency specified by CURRENCY.
SHIPTAXAMOUNT	DECIMAL (20,5) NULL	The total shipping taxes associated with this OrderItem, in the Currency specified by CURRENCY.
ESTAVAILTIME	TIMESTAMP NULL	An estimate of when sufficient inventory will be available to fulfill this OrderItem. This estimate does not include the shipping offset.
FIELD1	INTEGER NULL	Customizable.
DESCRIPTION	VARCHAR (254) NULL	A mnemonic description of the OrderItem, suitable for display to the customer. This field is usually NULL when CATENTRY_ID is not NULL, since in that case the CatalogEntry description can be displayed.
FIELD2	VARCHAR (254) NULL	Customizable.
ALLOCATIONGROUP	BIGINT NULL	Reserved for IBM internal use.
SHIPCHARGE	DECIMAL (20,5) NULL	The shipping charge associated with this OrderItem, in the currency specified by CURRENCY.
BASEPRICE	DECIMAL (20,5) NULL	If PRICE was converted from a currency different from the OrderItem currency, BASEPRICE is the price that was converted to arrive at PRICE.
BASECURRENCY	CHARACTER (3) NULL	The currency of BASEPRICE.
TRACKNUMBER	VARCHAR (64) NULL	Reserved for IBM internal use.
TRACKDATE	TIMESTAMP NULL	Reserved for IBM internal use.
PREPAREFLAGS	INTEGER NOT NULL,	Contains the following bit flags indicating special processing to be performed by the

	DEFAULT 0	OrderPrepare command: 1 = generated - the OrderItem was generated during a previous execution of the OrderPrepare command. The next time the OrderPrepare command is run, it first removes all generated OrderItems, so they can be re-generated if and as applicable. 2 = priceOverride - the price of the OrderItem has been manually entered, and will not be changed by customer commands. 4 = fulfillmentCenterOverride - the FulfillmentCenter has been manually specified, and will not be changed by customer commands. 8 = directCalculationCodeAttachment - CalculationCodes may be directly attached to the OrderItem. The default CalculationCodeCombineMethod will not look for direct attachments unless this flag is true. 16 = shippingChargeByCarrier - The contract for this OrderItem indicates that no shipping charges will be calculated by WebSphere Commerce. It may be calculated and charged by the carrier upon fulfillment. 32 = quotation - the OrderItem was obtained from a quotation. The price will not be automatically refreshed by customer commands. 64 = notConfigured - Price lookup and inventory allocation for this OrderItem should not be done using the component items found in the OICOMPLIST table for this OrderItem. For backward compatibility, this flag does not need to be set for OrderItems whose configurationId column value is null. 128 = autoAdd - OrderItem was added
CORRELATIONGROUP	BIGINT NULL	Normally this is the same as ORDERITEMS_ID, except: 1. when an OrderItem is split by the AllocateInventory task command, the newly created OrderItem inherits the CORRELATIONGROUP value from the original OrderItem. 2. when the PREPAREFLAGS column indicates

		"quotation", the OrderItem inherits the CORRELATIONGROUP value from the corresponding OrderItem in the parent Order.
PROMISEDAVAILTIME	TIMESTAMP NULL	When an Order is placed (using the OrderProcess command), this would be set to EstAvailTime. After that it would normally not be updated, although a CSR could manually update this to reflect a verbal commitment made to the customer.
SHIPPINGOFFSET	INTEGER NOT NULL, DEFAULT 0	An estimate of how many seconds it will take to ship this item once the Order is placed and inventory has been allocated.
NEEDEDQUANTITY	INTEGER NOT NULL, DEFAULT 0	Quantity needed for fulfillment. If CATENTRY is not NULL, this is QUANTITY times CATENTSHIP.NOMINALQUANTITY, converted from CATENTSHIP.QUANTITYMEASURE to BASEITEM.QUANTITYMEASURE, divided by BASEITEM.QUANTITYMULTIPLE and rounded to the nearest integer.
ALLOCQUANTITY	INTEGER NOT NULL, DEFAULT 0	Quantity allocated or backordered for this OrderItem. The quantity in BASEITEM.QUANTITYMEASURE units can be calculated by multiplying this value by BASEITEM.QUANTITYMULTIPLE, for the BaseItem of the specified item indicated by ITEMSPC_ID.
ALLOCFFMC_ID	INTEGER NULL	The FulfillmentCenter from which inventory for this OrderItem is allocated or backordered.
ORDRELEASENUM	INTEGER NULL	The associated OrderRelease, if any.
CONFIGURATIONID	VARCHAR (128) NULL	The identifier that is provided by an external product configurator. This identifier represents a list of order item components that are stored in the OICOMPLIST table.
SUPPLIERDATA	VARCHAR (254) NULL	Opaque to WebSphere Commerce. This attribute can be returned with a quotation, and sent when an Order is placed on an external system. For example, it could contain a supplier distribution center ID.
SUPPLIERPARTNUMBER	VARCHAR (254) NULL	The supplier part number, if known. Suitable for display to the customer.
AVAILQUANTITY	INTEGER NULL	If specified, indicates the quantity available for purchase.

APPENDIX 5. WebSphere Commerce database table: ORDERS

Each row in this table represents an Order in a Store.

Column Name	Column Type	Column Description
ORDERS_ID	BIGINT NOT NULL	Generated unique key.
ORMORDER	CHARACTER (30) NULL	A merchant assigned order reference number, if any.
ORGENTITY_ID	BIGINT NULL	The immediate parent organization ID of the creator.
TOTALPRODUCT	DECIMAL (20,5) NULL, DEFAULT 0	The sum of ORDERITEMS.TOTALPRODUCT for the OrderItems in the Order.
TOTALTAX	DECIMAL (20,5) NULL, DEFAULT 0	The sum of ORDERITEMS.TAXAMOUNT for the OrderItems in the Order.
TOTALSHIPPING	DECIMAL (20,5) NULL, DEFAULT 0	The sum of ORDERITEMS.SHIPCHARGE for the OrderItems in the Order.
TOTALTAXSHIPPING	DECIMAL (20,5) NULL, DEFAULT 0	The sum of ORDERITEMS.SHIPTAXAMOUNT for the OrderItems in the Order.
DESCRIPTION	VARCHAR (254) NULL	A mnemonic description of the Order, entered by the customer, suitable for display to the customer.
STOREENT_ID	INTEGER NOT NULL	The StoreEntity the Order is part of. This is normally a Store unless STATUS is Q, in which case it is normally a StoreGroup.
CURRENCY	CHARACTER (10) NULL	The Currency for monetary amounts associated with this Order. This is a currency code as per ISO 4217 standards.
LOCKED	CHARACTER (1) NULL	Reserved for IBM internal use.
TIMEPLACED	TIMESTAMP NULL	The time this Order was processed by the OrderProcess command.
LASTUPDATE	TIMESTAMP NULL	The time this Order was most recently updated.
SEQUENCE	DOUBLE NOT NULL, DEFAULT 0	May be used by a user interface to control the sequence of Orders in a list.
STATUS	CHARACTER (1) NULL	The status of the Order.
MEMBER_ID	BIGINT NOT NULL	The customer of the Order.

FIELD1	INTEGER NULL	Customizable.
ADDRESS_ID	BIGINT NULL	The billing address, if known.
FIELD2	DECIMAL (20,5) NULL	Customizable.
PROVIDERORDERNUM	INTEGER NULL	Reserved for IBM internal use.
SHIPASCOMPLETE	CHARACTER (1) NOT NULL, DEFAULT 'Y'	Reserved for IBM internal use.
FIELD3	VARCHAR (254) NULL	Customizable.
TOTALADJUSTMENT	DECIMAL (20,5) NULL, DEFAULT 0	The sum of ORDERITEMS.TOTALADJUSTMENT for the OrderItems in the Order.
ORDCHNLTYP_ID	BIGINT NULL	Reserved for IBM internal use.
COMMENTS	VARCHAR (254) NULL	Comments from the customer.
NOTIFICATIONID	BIGINT NULL	Notification identifier referring to the rows in the NOTIFY table that store notification attributes. These attributes override the defaults for notifications related to this order.

APPENDIX 6. WebSphere Commerce database table: PX_CDPOOL

This table contains all the promotion codes available to shoppers.

Column Name	Column Type	Column Description
PX_CDPOOL_ID	BIGINT NOT NULL	Primary key.
STORE_ID	INTEGER NOT NULL	Foreign key to STORE.
USAGETYPE	SMALLINT NOT NULL DEFAULT 0	0: Private, single use 1: Public, multiple uses
CODE	VARCHAR(128) NOT NULL	The code string.
STATUS	SMALLINT NOT NULL	0: Inactive 1: Active 2: Mark for delete
TRANSFERABLE	SMALLINT NOT NULL DEFAULT 0	0: Transferring allowed 1: Transferring not allowed
VALIDFROM	TIMESTAMP	Currently not in use.
VALIDUNTIL	TIMESTAMP	Currently not in use.
OPTCOUNTER	SMALLINT NOT NULL DEFAULT 0	Reserved for IBM internal use.

APPENDIX 7. WebSphere Commerce database table: PX_CDPROMO

This table contains all the promotion code information for relationships with a promotion.

Column Name	Column Type	Column Description
PX_CDPROMO_ID	BIGINT NOT NULL	Primary key.
PX_CDPOOL_ID	BIGINT NOT NULL	Foreign key to PX_CDPOOL.
PX_PROMOTION_ID	INTEGER NOT NULL	Foreign key to PX_PROMOTION.
STATUS	SMALLINT NOT NULL DEFAULT 0	0: Inactive 1: Active
LASTUPDATE	TIMESTAMP	Currently not in use.
OPTCOUNTER	SMALLINT NOT NULL DEFAULT 0	Reserved for IBM internal use.

APPENDIX 8. WebSphere Commerce database table: PX_PROMOTION

This table contains promotions.

Column Name	Column Type	Column Description
PX_PROMOTION_ID	INTEGER NOT NULL	The promotion ID. This is a primary key.
PRIORITY	INTEGER NOT NULL	The priority of the promotion.
STATUS	INTEGER NOT NULL	The current status of a promotion. Possible values include: 0: inactive 1: active 2: marked for deletion 3: suspended 4: obsolete 5: activating
EXCLSVE	INTEGER NOT NULL	The exclusivity of a promotion. Possible values include: 0: This promotion may be combined with any other promotions 1: This promotion can not be combined with any other promotions in the same promotion group 2: This promotion can not be combined with any other promotions
TYPE	INTEGER NOT	Contains the type of the promotion.

	NULL	Possible values include: 0: This promotion is applicable to people who belong to one or more of the targeted customer profiles. When the targeted profile file list is empty, it applies to everyone 1: This promotion is applicable only to those to whom it has been explicitly granted, that is, this promotion is a coupon promotion
PERORDLMT	INTEGER NOT NULL DEFAULT -1	The number of times this promotion can be applied to a single order. A value of -1, which is the default, imposes no limit.
PERSHOPPERLMT	INTEGER NOT NULL DEFAULT -1	The number of times a customer can redeem this promotion. A value of -1, which is the default, imposes no limit.
TOTALLMT	INTEGER NOT NULL DEFAULT -1	The overall limit on the number of times this promotion can be redeemed. A value of -1, which is the default, imposes no limit.
RSV_INT	INTEGER	Reserved for IBM internal use.
PX_GROUP_ID	INTEGER NOT NULL	Foreign key to the PX_GROUP table. This identifies the promotion group to which this promotion belongs.
CAMPAIGN_ID	INTEGER	Foreign key to CAMPAIGN table. This identifies the campaign with which this promotion is associated. This field is intended for customization and not used by default.
STOREENT_ID	INTEGER NOT NULL	Foreign key to the STOREENT table. This identifies the store to which this promotion belongs.
VERSION	INTEGER NOT NULL	The version number of this promotion.
REVISION	INTEGER NOT NULL	The version number of this promotion.
EFFECTIVE	INTEGER	If this promotion is a private promotion, this column defines how many days after the coupon is issued the coupon becomes active.
TRANSFER	INTEGER DEFAULT 0	If this promotion is a private promotion, the value in this column deterimines whether the coupons generated for this promotion are transferrable or not. A value of 0, which is the default, does not allow the coupon to be transferred among customers.
CDREQUIRED	INTEGER NOT	A flag that determines whether a promotion

	NULL DEFAULT 0	code is required to redeem this promotion. A value of 0, which is the default, means that a code is not required.
EXPIRE	INTEGER	If this promotion is a private promotion, this column defines how many days, after the coupon becomes active, it expires.
LASTUPDATEBY	BIGINT	The ID of the user that last modified the promotion.
LASTUPDATE	TIMESTAMP	The timestamp of the most recent update to this promotion.
STARTDATE	TIMESTAMP NOT NULL	The overall starting date for this promotion.
ENDDATE	TIMESTAMP NOT NULL	The overall end date for this promotion.
RSV_TIME	TIMESTAMP	Reserved for IBM internal use.
RSV_REAL	DECIMAL (20,5)	Reserved for IBM internal use.
TGTSALES	DECIMAL (20,5)	The target sales figure for this promotion.
NAME	VARCHAR(128) NOT NULL	The name of the promotion.
CODE	VARCHAR(128)	The promotion code for this promotion.
RSV_VCH	VARCHAR(254)	Reserved for IBM internal use.
XMLPARAM	CLOB	The XML definition of this promotion.
OPTCOUNTER	SMALLINT	Reserved for IBM internal use.

APPENDIX 9. WebSphere Commerce database table: USERDEMO

This table stores demographics information for users.

Column Name	Column Type	Column Description
USERS_ID	BIGINT NOT NULL	Foreign key to the USER table for the member (user) who is associated with this demographic information.
GENDER	CHAR(1)	The gender of the user. Valid values are as follows: F = female M = male N = not provided If not provided, N is used as the default.
AGE	INTEGER	The age of the user. The default value is 0 (age not provided). Other values are configurable.
INCOME	INTEGER	The annual income for the user. Valid values are configurable.
MARITALSTATUS	CHAR(1)	The marital status of the user. Valid values are configurable.
INCOMECURRENCY	CHAR(3)	Currency for the income of the user.
CHILDREN	INTEGER	The number of children that the user has. If not provided, the default is 0.
HOUSEHOLD	INTEGER	Number of people in the household.
COMPANYNAME	VARCHAR(128)	The company for which the user works.
HOBBIES	VARCHAR(254)	The main interests and hobbies of the user.
ORDERBEFORE	CHAR(1)	Indicates whether or not the user has previously placed an order. This value is supplied by the user.
FIELD1	CHAR(1)	Customizable.
TIMEZONE	CHAR(5)	The time zone in which the user resides.
FIELD2	CHAR(1)	Customizable.
FIELD7	VARCHAR(64)	Customizable.
FIELD3	CHAR(1)	Customizable.
FIELD4	CHAR(1)	Customizable.
FIELD5	VARCHAR(254)	Customizable.
FIELD6	INTEGER	Customizable.
DATEOFBIRTH	DATE	The date of birth of the user stored as an absolute point in time. The year 1896 is internally designated as the year if the user does not provide a year of birth, as it includes all valid dates in the Gregorian calendar.
OPTCOUNTER	SMALLINT	Reserved for IBM internal use.

APPENDIX 10. WebSphere Commerce database table: USERS

This table contains all users of the WebSphere Commerce system: registered users, guest users, and generic users.

Column Name	Column Type	Column Description
USERS_ID	BIGINT NOT NULL	ID for the user member. Foreign key to MEMBER_ID in MEMBER table.
DN	VARCHAR(1000)	The distinguished name of the user, for example, uid=wcsadmin,o=root organization. It must be in lower case, and should not have any spaces immediately before or after the , or = symbols. When using LDAP, this value must correspond with the DN of the user in the LDAP server.
REGISTERTYPE	CHAR(4) NOT NULL	The user registration type. Valid values are as follows: R - registered user G - guest user A - administrator S - site administrator The default member group called Administrators defines the list of administrative roles.
PROFILETYPE	CHAR(2)	Identifies whether the user has a profile, and if so, the profile type. Valid values are as follows: Null - no profile data C - base profile data B - business profile data
LANGUAGE_ID	INTEGER	Preferred language. For a list of language components, see the LANGUAGE table. Foreign key relationship to LANGUAGE table.
FIELD1	VARCHAR(254)	Customizable.
SETCCURR	CHAR(3)	Preferred currency in 3-character alphabetic code as per ISO 4217. This is a currency code as per ISO 4217 standards. Compare with the SHPREFERREDCURR column of the SHOPPER table provided with previous versions of WebSphere Commerce or WebSphere Commerce Suite.

FIELD3	VARCHAR(254)	Customizable.
FIELD2	VARCHAR(254)	Customizable.
LASTORDER	TIMESTAMP	The date and time that the user last placed an order at this site.
REGISTRATION	TIMESTAMP	The date or time that the user was registered, directly by way of UserRegistrationAdd, during synchronization from LDAP to the WebSphere Commerce database.
LASTSESSION	TIMESTAMP	The date and time that the user last visited the WebSphere Commerce site. Last visited means last logon to the WebSphere Commerce site.
REGISTRATIONUPDATE	TIMESTAMP	The date or time the user last changed registration information. This value is set during UserRegistrationAdd and UserRegistrationUpdate ResetPassword synchronization with LDAP during logon.
REGISTRATIONCANCEL	TIMESTAMP	Reserved for IBM internal use.
PREVLASTSESSION	TIMESTAMP	Reserved for IBM internal use.
OPTCOUNTER	SMALLINT	Reserved for IBM internal use.
PERSONALIZATIONID	VARCHAR(30)	The Personalization ID associated with the user